Routing problem

An implementation of AStar algorithm

Joseba Hernández Bravo and Jorge Vicente Puig

Universitat Autònoma
de Barcelona

November 2021

Contents

1 Routing problem

In this introductory first section we will introduce the meaning of a routing problem along with the specific problem to solve and the algorithm used for it.

The routing problem, more specific the network routing problem, consist of finding an optimum route between two or more nodes in relation to total time, cost or distance. The edges, that can be oneway or twoways, are weighted by some cost to traverse them. Also, it can exist many prohibitions, such as only visiting the nodes once.

This problem arise naturally in many different fields, such as transport or communications.

1.1 Problem description

The problem to solve consists in computing the optimal path (i.e. the path that minimise distance) from *Basílica de Santa Maria del Mar* a place located in Barcelona to the *Giralda* in Sevilla.
The data used for solving the problem consist of a *csv file* with a list of:

- **Nodes**: correspond to point locations on the earth. We will use the next information of it: a unique *id*, the *latitude* and the *longitude* of the node.
- **Ways**: correspond to the edges between the nodes. We will use the next information of it: the way *name*, a *oneway* parameter that indicates if it is a oneway or twoway road and a list of nodes that are in that way.
- **Relations**: is a list of characteristics that are irrelevant for our problem so we will not use them.

All of this information will be used to construct a weighted directed graph composed by the nodes and ways. Also, we can notice that our problem doesn't have any extra prohibition.

Figure 1: An example of a graph on Spain

On this graph we consider a *source* vertex and a *goal* vertex and by applying a shortest path algorithm, in our case the A^*, we will be able to compute the minimum distance between these nodes.

1.2 AStar algorithm

Let's consider a strongly positive directed weighted graph $G = (V, E, \omega)$ where V is our nodes set, E is our edge set and ω is an edge-weight function:

$$\begin{array}{rccc} \omega: & E & \longrightarrow & \mathbb{R} \\ & a & \longmapsto & \omega(a) \end{array}$$

Let's consider as source node $s \in V$ and as goal node $g \in V$. Then, our objective is to find a minimal path from s to g, i.e. the path α that minimises

$$\omega(\alpha) = \sum_{i=0}^{n} \omega((v_i, v_{i+1}))$$

where $v_0 = s$, $v_N = g$ and $v_i \in V$ for all $0 \leq i \leq N$.

This problem can be solved with many different algorithms, for instance the *Dijkstra's Algorithm*, however we will use the A^* *Algorithm* that has better performance. The A^* *Algorithm* is a complete, optimal and computationally efficient algorithm, in fact, is the best solution in many cases.

The key point of the algorithm is the use of an **heuristic function**. In fact, at each iteration of the main loop of the algorithm we have to choose

the best path to follow.

In order to do that, we will not only choose the path from w to v based on the weight of the path, we will also include an heuristic function $h(v)$ such that:

$$f(v) = \omega(v, w) + h(v)$$

where $f(v)$ will be the total cost. And our algorithm will choose the node v that minimises $f(v)$.
Notice that the heuristic function will return a distance from the node v to the goal node g. A deeper discussion of the heuristic function will be done in Subsection 2.2 .

A more detailed explanation of the *AStar algorithm* is done in Subsection 2.3 where it is discussed the implementation.

2 Code explanation and main ideas

In this section we will explain all the important ideas used while implementing the code of the program. We will also include fragments of code that will show how the ideas are implemented.

2.1 Structure used

In order to choose an appropriate structure to use in the program we will need to store: the id, the latitude, the longitude and the number of successors. We also need to store the successors of each node and the the name of the way that connect the node to each successor.

A first approach would be an structure like this:

```
typedef struct{
    unsigned long id;
    char name[20];
    double latitude,longitude;
    unsigned short numbersegments;
    unsigned long segment[9];
}node;
```

However, that is a really useless way of storing the information. In fact, there are many names of ways that doesn't exists. Also, the average valence of the nodes is 1.9333, so it would be many empty successors.

Therefore, we need to make a dynamic storage of the names and of the successors:

```
typedef struct{
    unsigned long id;
    char *name;
    double latitude,longitude;
    unsigned short numbersegments;
    unsigned long *segment;
}node;
```

The implementation of the dynamic storage will be done firstly initialising the *names* and *segments* using the *malloc* function and then, while reading the information stored on the ways we will add more space if needed using the *realloc* function.

Remark: The construction of the graph from the *csv* file will be done in a different script and save it in a binary file, in order to have a faster implementation of the A^* algorithm while reading the graph.

2.2 Heuristic function

In this subsection we will discuss the heuristic function along with different examples and implementations.

The heuristic function $h(v)$ tells to the A^* an **estimate** of the minimum cost from any node $v \in V$ to the goal. In this way it takes the highest value at the initial node and decreases to a value of 0 at the final node. If we have that $h(v) = 0 \ \forall v \in V$ the A^* is equivalent to the Dijkstra's algorithm.

Choosing an appropriate heuristic will play a major role on the algorithm. The heuristic function should hold next properties:

- **Admissibility:** An heuristic function h is considered to be admissible if $\forall v \in V$, $h(v) \leq \omega(v, g)$ where g is the goal node. That means, the heuristic is always lower than the cost from moving from v to g.

  In fact, if this property is verified, A^* is guaranteed to find an optimal path. And in the case where $h(v) = \omega(v, g)$, A^* will have perfect performance.

- **Consistency:** A heuristic function h is considered to be consistent if it satisfies the triangular inequality. Thus, for each pair of vertexes $x, y \in V$ we have that $h(x) \leq d(x, y) + h(y)$.

  An important property is that every consistent heuristic is admissible. Thus, is enough to find an heuristic that holds triangular inequality for having an A^* algorithm that returns the optimal path.

Another important point to take care is the speed vs accuracy of the A^*. In fact, the behaviour of the A^* is based on the heuristic function, if we

choose an admissible heuristic that approximates ω we will be sure of finding the optimal path, however it might too many time.

By setting an heuristic that is not admissible we can speed up the algorithm and obtain a path that is close enough to the optimal one.

We will look at different results using three heuristics:

- **Null heuristic.** Is simply the $h(v) = 0 \ \forall v \in V$, i.e., the Dijkstra's algorithm.
- **Straight distance.** It computes the straight distance from any node $v \in V$ to the goal node.
- **Successors way.** It return the minimum distance of the path from the node to a successor and the straight to the goal node.

Results of the heuristics are presented in the following table:

Time in seconds and number of iterations for different heuristics

Null heuristic	Straight distance	Successors way
46.58	12.78	15.11
12321359	2 850 343	2870434

Looking at this results we have decided to use the *Straight distance* as the heuristic for Sections 3 and 4.

2.3 AStar implementation

Before discussing the implementation of the A^* algorithm we need to decide how we will measure the distance between two nodes.

As known, the distance between two points on the earth is not well defined and then there are many possible distances to choose. We will focus on three of them:

- **Equirectangular approximation:** Is based on the equirectangular projection that is a cylindrical geographical projection. It projects the

sphere into a plane.

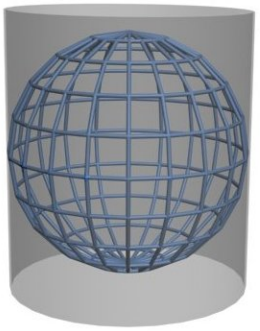

Figure 2: Cylindrical Projection

Then, we are transforming the spherical coordinates into planar coordinates by the simple transformation:

$$\begin{cases} x &= (\Delta\lambda)cos(\frac{\phi_1+\phi_2}{2}) \\ y &= (\Delta\phi) \end{cases} \tag{2.1}$$

where ϕ_i represents the longitude, $\Delta\phi = \phi_2 - \phi_1$ the difference on longitude and $\Delta\lambda = \lambda_2 - \lambda_1$ the difference on latitude.

Then the Pythagoras's theorem will be used for computing the distance. This distance give us a really quick way of computing the distance, however is not an accurate distance.

- **Spherical Law of Cosines:** This law consists on a theorem of differential geometry that relates the sides and angles of spherical triangles.

  Given the lengths of the three sides a, b and c and the angle C adjacent to a and b, then the spherical law of cosines states that:

$$cos(c) = cos(a)cos(b) + sin(a)sin(b)cos(C) \tag{2.2}$$

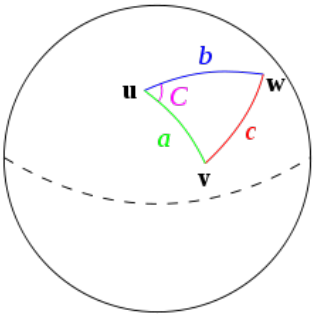

Figure 3: Spherical Triangle. Image from: [2].

In our case, rearranging the canonical form we can directly use the latitude and longitude by the following equation:

$$cos(dist) = sin(\lambda_1)sin(\lambda_2) + cos(\lambda_1)cos(\lambda_2)cos(\Delta\phi) \tag{2.3}$$

where λ_i represent the latitude and $\Delta\phi = \phi_2 - \phi_1$ the difference on longitude.

The law of spherical cosines give us an accurate and not very expensive way of computing the distances between two points. However, this law is ill-conditioned for small distances due to computer arithmetic when computing *arccos* function.

- **Haversine formula:** This formula determines the distance between two points in a sphere given their latitudes and longitudes. It comes from the *Law of Haversine* that is a general case of the *Spherical Law of Cosines* (in fact, this Law is deduced from it).

  The *Haversine formula* states that,

$$sin^2\left(\frac{dist}{2Radius}\right) = sin^2\left(\frac{\Delta\lambda}{2}\right) + cos(\lambda_1)cos(\lambda_2)sin^2\left(\frac{\Delta\phi}{2}\right) \tag{2.4}$$

  where λ_i represent the latitude, ϕ_i the longitude, $\Delta\lambda = \lambda_2 - \lambda_1$ and $\Delta\phi = \phi_2 - \phi_1$ the respective differences.

The Haversine formula gives us a really accurate way of measuring distance between two points in earth, even for small distances. However, it is more computationally expensive than the other distances.

Remark: For simplicity in all distances has been considered 6371km as a constant for the earth radio.

The distance used to compute the weights between nodes will be the *Spherical Law of Cosines*, as the *Haversine formula* is computationally expensive and the *Equirectangular projection* has low accuracy.

At this point we can explain the implementation of the A^*.

Firstly, in order to store the path in an organised and computationally efficient way, we have made use of an *open queue*. This structure contains the following functions:

- Add with priority: add a new non-existing element to the queue and sort it according to the value $f = h + g$. The lower the value of f the further to the left the new element will be placed.
- Requeue with priority: sort an existing element according to the value $f = h + g$. As in the previous case, the lower the value of f the further to the left the new element will be placed.
- Extract min: Takes the first element of the queue (the one in the left limit).

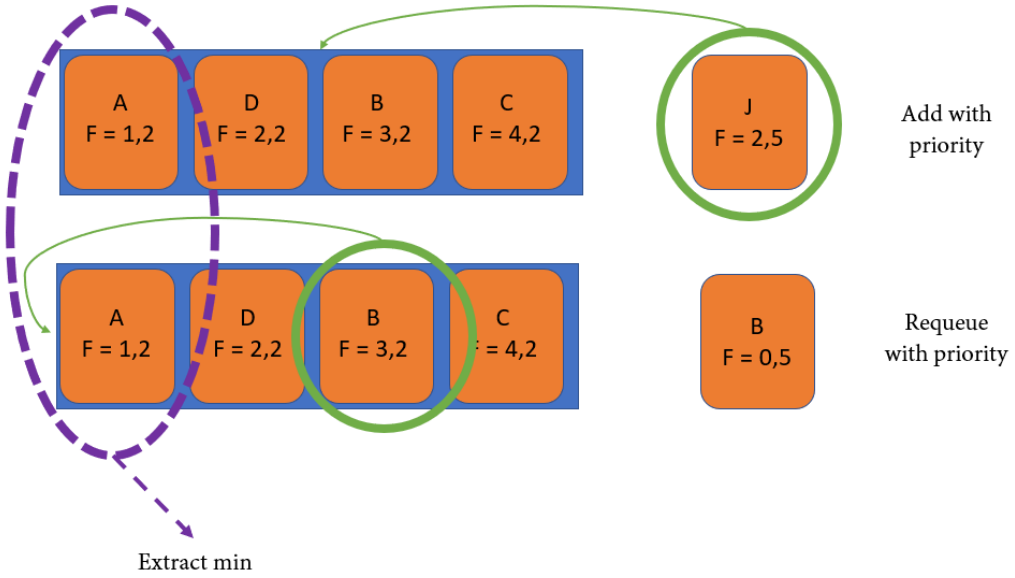

Figure 4: Different open queue functions for a given new element.

Then, the iterations of the algorithm are done with the next structure:

```
while OpenQueue is not empty
    /* Take out first node and check if is the goal node */
    if((node = extract_min(PathData)) == node_goal) return true;
    /* Looking for the best path to follow */
    for each successor of node
        f_successor = g_node + omega(successor, node)
        /* Relaxation step */
        if (f_successor < g_successor)
            /* Better path found */
            parent_successor = node
            g_successor = f_successor
            /* Adding to the path data */
            if (successor belongs to PathData) Requeue_with_priority(successor)
            else Add_with_priority(successor)
```

Remark: All the code can be found on the next *GitHub* repository: https://github.com/joseb061/Master-projects/tree/master/Optimization-Astar.

3 Results obtained

In this section we will analyse the results obtained from the A^* algorithm. Also, we will apply the following performance strategy:

- **Weighting the heuristic function.** As explained in section 2.2 the heuristic function gets optimal when $h(v) = \omega(v, g)$, so we will rerun our algorithm with the next cost function:

  $$f(v) = \omega(v, w) + \lambda h(v), \qquad \lambda \in \mathcal{I}$$

  where λ is the weight parameter in some real interval $\mathcal{I}$.

  In our case we will set $\mathcal{I} = [0.8, 1.2]$, and discretise it in 101 values, i.e. we will compute:

  $$f(v) = \omega(v, w) + (0.8 + k\frac{0.4}{100})h(v), \qquad k = 0, \ldots, 100$$

3.1 Weighting the heuristic function

Doing the performance improvement explained before, we obtain the next results, explained in the next graphics where we compare the distance, iterations and execution time depending on the value of lambda:

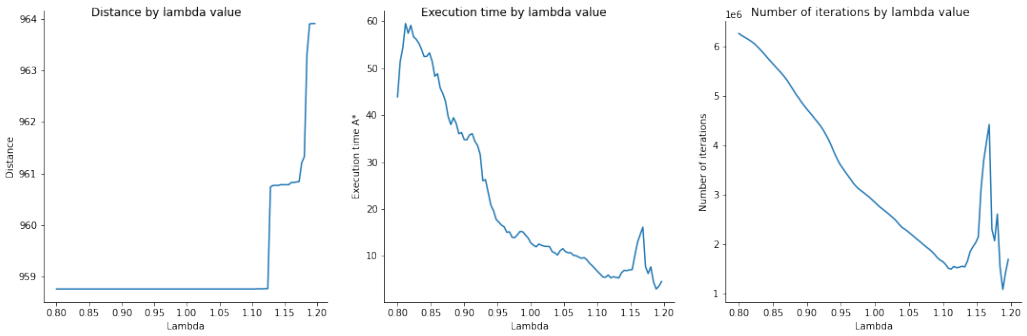

Figure 5: Distance, execution time and nº of iterations

Looking at this graphic seems to have a proper value of lambda to minimise execution time while having the optimal path. We can see a better comparison in the next graphic:

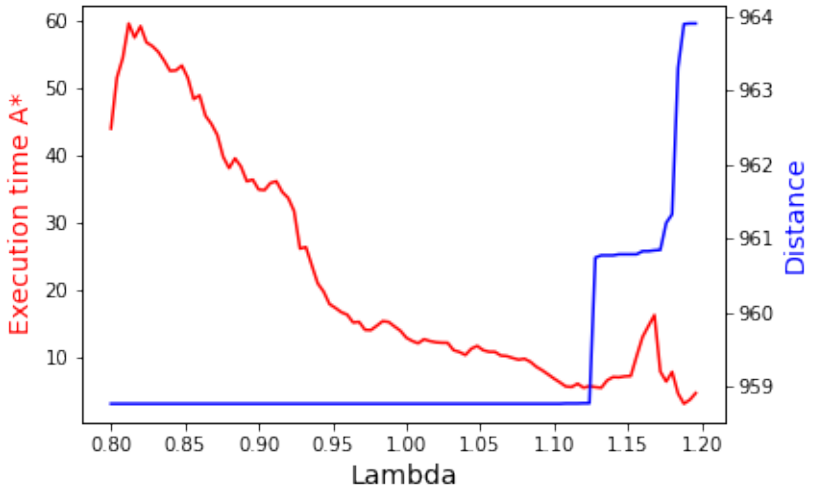

Figure 6: Distance and execution time depending on λ value

Remark: All graphics have been made *Matplotlib* and *Seaborn* libraries in *Python*.

Then, we can clearly appreciate a good performance improvement. In fact, by setting $\lambda = 1.120$, we can observe that the execution time is reduced from 12.78 seconds to 5.35 seconds at the same time as the optimal path is found.

In fact, as explained in Subsection 2.2 by scaling the heuristic, we are making a more accurate estimation of the real distance which results in a reduction of the execution time.

Remark: The characteristics of the computer on which we run the programmes are:

- Operating system: Ubuntu 20.04.3 LTS.
- Processor: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz.
- RAM: 8GiB.

3.2 Optimal path

Once we run our programme, we obtain two types of output files.

On the one hand, a file[1] with the characteristics of each node of the path separated by the symbol "|". This file contains the following information:

- The Id of the current node.
- The distance travelled from the starting point to the current node.
- The latitude of the current node.
- The longitude of the current node.
- A sequence of the streets passing through the current node.

On the other hand, the program exports a second file[2] with all the latitudes and longitudes of each node of the route. This file has been exported with the extension "*.csv*" precisely to be plotted with the platform **Google earth pro**.

Figure 7: Optimal route from from *Basilica de Santa Maria del Mar* to the *Giralda* .

[1]Link to the results: https://github.com/joseb061/Master-projects/blob/master/Optimization-Astar/main_algorithm_SPAIN/RESTULTS.txt.

[2]Link to the data used in **Google earth pro**: https://github.com/joseb061/Master-projects/blob/master/Optimization-Astar/main_algorithm_SPAIN/for_maps/pathsolution.csv.

3.3 Conclusions

In this work we have used the AStar algorithm to calculate the optimal route from the *Basílica de Santa Maria del Mar* in Barcelona to the *Giralda* in Sevilla. It should be notice that what has been developed by means of the algorithm in this work is the optimisation in distance and not in time.

The work has been divided into two parts. In the first part we have processed and structured all the available the data using the scheme explained in the Subsection 2.1. Once the data was structured, we exported them in a binary file. In this way, we have managed to have the map in the form of a graph.
In the second part, using the binary file and defining a starting and a final node, we have found the minimum path between these two points using the AStar algorithm.

In the second part of the project we have made several decisions. After testing 3 different heuristics we have concluded that using the **Straight distance** heuristic (Subsection 2.2) we achieved a shorter execution time compared to the other two heuristics. From this result we can deduce that the main reason why the heuristic is faster than the others is due to its simplicity. Firstly, the **null heuristic** has a much higher computational cost since the number of nodes visited is much higher. A more "sophisticated" heuristic, such as the **Successors way**, gives us a more accurate result although it requires more nodes to be visited. Therefore, the execution time is longer.

Furthermore, for the calculation of the distance between different nodes, we have decided to use the *Spherical Law of Cosines* as the *Haversine formula* is computationally expensive and the *Equirectangular projection* has low accuracy.

Finally, once we have chosen the correct heuristics and the way to measure the distances between nodes, we have tried to optimise the execution time of the algorithm in the Section 3. To do so, we have made use of the weight parameter λ, which has given us the best result for the value of 1.120. Using this parameter we have reduced the execution time from 12.78s to 5.35s.

References

[1] Lluís Alsedà. *A* algorithm pseudocode. Lecure notes in Optimization.* http://lluis-alseda.cat/MasterOpt/index.html

[2] Wikipedia contributors. (2021, 23 april). Spherical law of cosines. Wikipedia. https://en.wikipedia.org/wiki/Spherical_law_of_cosines

[3] *Calculate distance, bearing and more between Latitude/Longitude points* http://www.movable-type.co.uk/scripts/latlong.html